A Wait-free Algorithm for Optimistic Programming: HOPE Realized*

Crispin Cowan
Department of Computer Science and Engineering
Oregon Graduate Institute P.O. Box 91000
Portland, OR 97291-1000
crispin@cse.ogi.edu

Hanan L. Lutfiyya
Computer Science Department
University of Western Ontario
London, Ontario N6A 5B7
hanan@csd.uwo.ca

ABSTRACT

Optimism is a powerful technique for avoiding latency by increasing concurrency. Optimistically assuming the results of one computation allows other computations to execute in parallel, even when they depend on the assumed result. Optimistic techniques can particularly benefit distributed systems because of the critical impact of communications latency. This paper reviews HOPE: our model of optimistic programming, and describes how optimism can enhance distributed program performance by avoiding remote communications delay. We then present the wait-free algorithm used to implement HOPE in a distributed environment.

Keywords: optimism, concurrency, parallelism, distributed, rollback, wait-free, implementation.

1 INTRODUCTION

Optimism is a powerful technique for avoiding latency by increasing concurrency. Optimistically assuming the results of one computation allows other computations to execute in parallel, even when they depend on the assumed result. This paper describes an environment providing automatic assistance for writing optimistic programs, and presents the wait-free algorithm used to implement this environment.

Optimistic techniques can particularly benefit distributed systems. Communication latency is critical to performance because it limits the degree of parallelism. Optimism can mask communications latency by making optimistic assumptions about the behavior of remote nodes.

Avoiding communications latency is just a special case of the general technique of using optimistic assumptions to avoid latency by increasing concurrency. Optimism increases concurrency by making an assumption about a future state, and verifying the assumption in parallel with computations based on the optimistic assumption.

The classic example is optimistic concurrency control: assume that locks will be granted, process the transaction, and *post hoc* verify that the locks were granted [17]. This paper reviews how to avoid the latency of a remote procedure call by optimistically assuming that the call behaved as expected [1, 10, 11], illustrating why distributed systems particularly benefit from optimism: moving a computation to a remote node increases latency, but does not change the predictability of the computation.

Any assumption can be made, given a method to verify that the assumption is correct. If the assumption is discovered to be correct, then latency has been avoided and performance has improved. However, if the assumption is incorrect, then all computations that used the assumption must be rolled back and re-executed using correct data.

Optimism has been used to enhance performance in various areas [5, 20, 21], but mostly embedded in systems, and not exposed to the programmer. Optimism is rarely used in applications because optimistic programs are difficult to write and require ad hoc techniques to implement. *All* of the causal descendant computations of an optimistic assumption must be tracked, and rolled back if the assumption proves false; a process we call **dependency tracking**. Tracking all dependents of an optimistic assumption is tedious, at best, without automatic assistance.

We believe that optimism research has been hindered by the lack of adequate tools for making optimistic assumptions without worrying about the details of dependency tracking, checkpointing, and rollback. This paper presents HOPE (Hopefully Optimistic Programming Environment): a programming model for expressing optimism.

Previously, we defined the HOPE programming model and its applicability [5, 10]. We constructed a prototype HOPE system [6], defined a formal semantics for HOPE [9], and measured the performance of the prototype [11]. This paper presents the algorithm used to build the prototype. The algorithm is wait-free in that no user process ever blocks when executing a HOPE primitive. We also prove that some important properties of the prototype are consistent with HOPE's formal semantics.

Section 2 presents related work. Section 3 describes the HOPE programming model, Section 4 briefly describes the HOPE implementation, and Section 5 presents the wait-free algorithm used to construct the prototype. Section 6 presents our conclusions and future research.

2 RELATED WORK

Optimism has been used in a variety of applications such as fault tolerance [20, 15], replication [5, 16, 21], concurrency control [17], and discrete event simulation [2, 14]. However, optimism has not been generally exploited because of the difficulty in writing optimistic algorithms.

Optimistic programming is difficult, time-consuming, and *ad hoc*, for the following reasons. First, checkpointing and rollback is difficult and non-portable. Second, the programmer must keep track of all actions that must be rolled back if the assumption is

* Supported in part by the National Sciences and Engineering Research Council of Canada (NSERC) and ARPA.

wrong, including remote processes that have been sent messages. This requires that processes that make optimistic assumptions must keep track of all remote processes that messages were sent to (called **dependency tracking**). Third, the programmer must worry about **transitivity**; what if a computation that proceeds based on an optimistic assumption is used to decide whether another assumption is valid? If this optimistic computation is rolled back then the validation of the assumption must be undone.

Previous work in supporting optimistic programming includes [4, 14]. However, previous work has either restricted the type of optimistic assumption that can be made, or restricted the scope of optimistic computation [7]. In [4] computation based on an assumption is limited to the scope of a statically defined encapsulation. In Time Warp [14], the amount of computation based on a optimistic assumption is not statically bound, but only one kind of optimistic assumption can be made: that messages arrive in time-stamp order.

2.1 HOPE FEATURES

Any optimistic assumption can be made, and any method can be used to verify an optimistic assumption, including a method selected at run-time. HOPE is novel because we believe it to be the first system in which both the kind and scope of an optimistic assumption is unconstrained.

HOPE provides primitives that abstract the basic elements of optimism: specifying an optimistic assumption, identifying the assumption, and verifying correctness of the assumption. HOPE also provides the novel **assumption identifier** as a separate entity in the computational model. Previous optimistic systems have made optimistic computations directly dependent on other computations.

Optimistic assumptions can be affirmed or denied in parallel with ongoing computations that depend on the optimistic assumption. The primitives are general: any optimistic assumption can be made, and any user-programmed criteria can be used in deciding whether the optimistic assumption was correct. The verification criteria can also be selected at run time. Furthermore, affirmation and denial of optimistic assumptions, as well as making further optimistic assumptions, can all be performed by computations that are themselves optimistic, i.e. the primitives can be applied transitively. Despite the overhead required for these features, we have shown that HOPE can decrease the latency of a remote procedure call [11].

3 THE HOPE PROGRAMMING MODEL

The HOPE programming model is a set of primitives designed to be embedded in some other system. HOPE can be embedded in any message-based concurrent system.

Consider a distributed program composed of communicating sequential processes that execute operations that change the state of a process. A **computation** is a consecutive sequence of states in the execution of a process. **Rollback** returns a process to a previous state in its computation and discards subsequent computations. An **optimistic assumption** is an assertion about a future state that has yet to be verified. An **optimistic** or **speculative computation** is a one that proceeds based on an optimistic assumption and is **dependent** on that assumption. If the optimistic assumption is found to be true, then the optimistic computation is retained, otherwise it is rolled back. HOPE provides one data type and four primitives:

AID x	**x** is an **assumption identifier** or **AID**, used to identify particular optimistic assumptions.
guess(x)	Make an assumption identified by **x**. **guess** eagerly returns **True**, and returns **False** if rolled back.
affirm(x)	Assert that the optimistic assumption identified by **x** is true.
deny(x)	Assert that the optimistic assumption identified by **x** is false.
free_of(x)	Assert that the current computation is not dependent on the assumption identified by **x**.

An AID is a reference to an optimistic assumption which enables the primitives to separately specify dependence, precedence, and confirmation of an assumption.

guess(x) is a boolean function that returns **True** if the assumption identified by **x** is correct, and **False** if **x**'s assumption is found to be incorrect. **guess(x)** *eagerly* returns **True**, regardless of the status of the assumption; speculative computation begins at this point **dependent** on **x**. If **x**'s assumption is later discovered to be false, the process is rolled back to where it called **guess(x)**, and **False** is returned instead of **True**.

Idiomatically, **guess(x)** is embedded in an **if** statement. The "true" branch of the **if** statement contains the optimistic algorithm, and the "false" branch of the **if** statement contains the pessimistic algorithm. **aid_init(x)** is used to initialize **x** ahead of time, so that a checking mechanism can be set up to verify **x**'s assumption.

affirm(x) asserts that the assumption associated with the AID **x** is correct. **deny(x)** asserts that the assumption associated with **x** is incorrect. If **affirm(x)** is executed anywhere in the system, all the speculative computations executed from **guess(x)** onward are retained. If **deny(x)** is executed anywhere in the system, the computations from **guess(x)** onward are rolled back re-started from **guess(x)** with a return code of **False** instead of **True**.

There is no restriction on how much computation can be executed before an assumption is confirmed. Any process in the program may confirm an assumption. Only one **affirm** or **deny** primitive may be applied to a given assumption identifier, because multiple **affirm** or **deny** primitives are redundant, and conflicting **affirm** and **deny** primitives have no meaning. Speculative processes can execute **affirm** and **deny** primitives, and the system will transitively apply the assertions, i.e., if a speculative process is made definite, then all **affirm** primitives it has executed will have the same effect as definite **affirm** primitives.

In addition to explicit **guess** primitives, processes can also become dependent on AIDs by exchanging messages. A speculative process "tags" the messages it sends with the set of AIDs that it depends on. Receivers implicitly apply **guess** primitives to each of the AIDs in the message's tag.

free_of(x) asserts that the executing task is not dependent on the assumption identified by **x**. If such a dependency is detected, then **x** is denied, otherwise **x** is affirmed.

3.1 EXAMPLE: AVOIDING RPC DELAY

In a remote procedure call (RPC), the calling process is idle until it gets a response from the remote machine. Fast networks may not significantly reduce this idleness. For example, it takes 30 milliseconds to send a photon from New York to Los Angeles and back again. A transcontinental 100Mb/s network can send a 100 byte packets 100,000 times per second, but can only send that 100 byte packet 30 times per second if each transmission waits for a response. A 100 MIPS CPU can execute over 3 million instructions while waiting for a response from the opposite coast.

```
/* Worker Process */
line = call print("Total is ", total); /* S1 -- RPC */
if (line > PageSize)
   call newpage();                     /* S2 -- RPC */
call print("Summary ...");             /* S3 -- RPC */
/* ... end process */
```

Figure 1: Before Call Streaming Transformation

```
/* Worker Process */
aid_t    PartPage, Order;

PartPage = aid_init();
Order = aid_init();
send(WorryWart, PartPage, Order, total);
if (guess(PartPage))
   /* do nothing */                    /* S2 */
else
   call newpage();                     /* S2 -- RPC */
guess(Order);
call print("Summary ...");             /* S3 -- RPC */
/* ... end process. */


/* WorryWart Process(PartPage, total) */
aid_t    PartPage, Order;

receive(PartPage, Order, total);
line = call print("Total is ", total); /* S1 -- RPC */
free_of(Order);
if (line < PageSize)
   affirm(PartPage);
else
   deny(PartPage);
/* ... end process */
```

Figure 2: After Call Streaming Transformation

Let $S_1; S_2$ be two sequential RPC operations. Transforming the synchronous RPCs into asynchronous messages avoids RPC latency by executing S_1 and S_2 in parallel. If S_1 and S_2 are completely independent then it is easy to execute S_1 and S_2 concurrently. But what if S_1 and S_2 are not independent? The execution of S_2 may be a function of the response of the RPC done by S_1.

For example, Figure 1 shows a program fragment in which S_1 is an RPC that prints a summary total and returns the current line number of the page. S_2 takes the line number and checks to see if the line number now exceeds page size. If it does, then S_2 creates a new page; otherwise execution can immediately proceed to S_3.

Bacon and Strom's Call Streaming algorithm [1] optimistically parallelizes two such statements. We can parallelize S_1 and S_2 (and hence the statements after S_2) by making the optimistic assumption that the report does not end exactly at the bottom of the page, i.e., line < PageSize. Figure 2 shows how we can parallelize S_1 and S_2 as follows: S_1 is executed in the WorryWart process while S_2 (and the statements after S_2) is executed in the Worker process.

The Worker process spawns the WorryWart process to concurrently execute S_1. Worker then executes **guess(PartPage)**, and based on the optimistic **True** return code, executes S_2 and S_3 as if the line count were in fact less than PageSize, and prints "`Summary...`" without forcing a new page. However, the assumption has yet to be verified and the Worker computation is now speculative.

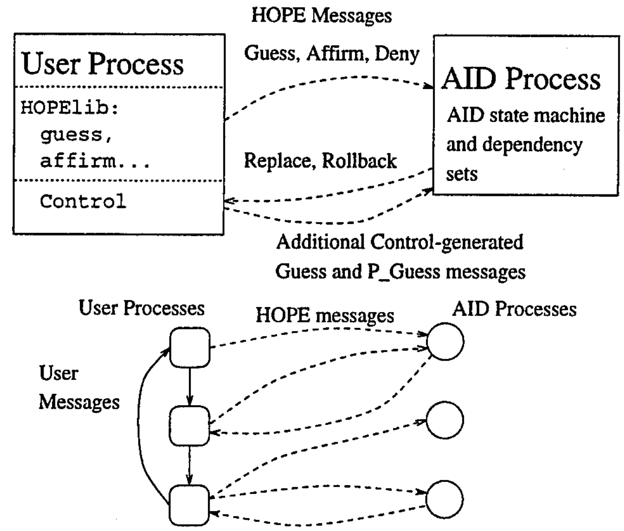

Figure 3: Structure of the Basic HOPE Model

If line < PageSize is not valid, then **deny(PartPage)** is executed, causing the Worker process to rollback to where it called **guess**. Any processes that Worker sent a message to while speculative are also rolled back. Worker now resumes execution with **False** returned from **guess(PartPage)**. This tells the Worker that the line value exceeded PageSize, so Worker calls newpage().

The execution of statements S_2 and onward must not interfere with S_1's execution. S_3's message may arrive at the remote machine ahead of the message from S_1 in the WorryWart process. The remote process becomes dependent on the assumption identifier **Order** and by transitivity the WorryWart process becomes dependent on **Order**. Because S_3 changes the line number, S_1's test is invalidated. The **free_of(Order)** primitive is used to detect this causality violation and force rollbacks to solve the problem.

4 PROTOTYPE SYSTEM

HOPE was built on top of a pre-existing message passing system to make optimistic techniques easily accessible. PVM [13] was selected because the source code was freely available, it is well supported by its authors, and has a broad user base, providing a potentially large audience for optimism. PVM is implemented as a library of message passing and administrative functions callable. The user's tasks run as ordinary processes on the host system.

Figure 3 shows the basic structure of the HOPE implementation. User programs access HOPE primitives through the `HOPElib` library, which contains functions for each of the HOPE primitives, as well as book-keeping functions that process HOPE messages for dependency tracking. Assumption identifiers are implemented as AID processes, which are spawned in the course of executing the HOPE `guess` function. AID processes track the set of processes that depend on the associated assumption identifier. [7] details the implementation, including techniques for checkpoint and rollback of a UNIX process.

5 THE HOPE ALGORITHM

The HOPE algorithm operates in an environment abstracted from the implementation in Section 4: concurrent processes that communicate via messages. The primary purpose of optimistic primitives is to avoid latency, so it is an important design criterion that all of the remote operations resulting from user processes executing HOPE primitives be asynchronous: user processes executing HOPE primitives should never have to wait for a message from another process. This section describes an algorithm to provide the HOPE primitives that is consistent with this design criterion, and uses only concurrent processes, messages, and a rollback facility for the processes.

As in Section 4, dependency tracking is implemented using a combination of AID processes and library functions attached to each user process. User process execution is recorded as an **execution history** of process states composed of **intervals**, as detailed in the formal semantics of HOPE [9]. An **interval** is a subsequence of an execution history between two executions of the **guess** primitive, and constitutes the smallest granularity of rollback that may occur. An interval is said to be **speculative** if that interval can be rolled back; otherwise, the interval is said to be **definite**. We use $A, B, \ldots$ to denote intervals in the history of user processes, $X, Y, \ldots$ to denote assumption identifiers (AIDs), and $P_X, P_Y, \ldots$ to denote AID *processes*.

The intervals in a user process's history, and the dependency tracking sets associated with each interval, are stored in data structures in the `HOPElib` attached to each user process, but hidden from the programmer. The key dependency set is the **IDO** (I Depend On) set of assumption identifiers that the interval depends on.

AID processes store and process dependency tracking information relating to the assumption that they identify. Local modifications to dependency sets are then simply local operations, and modifications to remote sets become messages requesting the modification to that set sent to the appropriate user or AID processes.

HOPE primitives are functions called from each user process. The functions make local modifications to the process history and local dependency sets, and then send messages to appropriate AID processes for further processing.

AID processes process messages from user processes, modifying their dependency tracking sets in response to their current state and the type and contents of the message. The AID processes compute the remaining dependency set and history changes, and send messages to other user processes. The messages sent to user processes are intercepted by the message passing system and given to the `HOPElib` attached to each user process for processing.

5.1 THE PROBLEM: INTERFERENCE

The HOPE operational semantics [9] specify the execution of a HOPE primitive as a sequence of operations. A direct implementation of these semantics in a distributed system requires updating variables in remote processes. Because HOPE primitives may be executed by concurrent processes, they are subject to concurrency errors due to interference [3, page 11], as detailed in [7].

One way to avoid interference problems is to prevent the interleaved execution of HOPE primitives by serializing execution of HOPE primitives, with unfortunate consequences. A more scalable approach would be to use some form of concurrency control on the HOPE data structures, producing a serializable execution of the HOPE primitives [3]. However, the time and space requirements of incorporating a general concurrency control system within the HOPE run-time are prohibitive. The remote communications latency inherent in a concurrency control protocol is precisely the form of delay HOPE was designed to avoid.

Type	From	To	Arguments	Meaning
Guess	User	AID	iid	Sender guesses AID is true
Affirm	User	AID	iid, IDO	Sender affirms AID, subject to IDO
Deny	User or AID	AID		Sender denies AID unconditionally
Replace	AID	User	iid, IDO	Replace sender with IDO in $iid.IDO$,
Rollback	AID	User	iid	Rollback interval iid

Table 1: Basic HOPE Messages

Rather than try to avoid conflicts, we allow conflicts to occur and correct after the fact using forward error recovery. Like concurrency control, this produces only serializable executions, but does so at a lower cost by exploiting specific knowledge of the HOPE primitives. We show that such an algorithm is consistent with the HOPE semantics by showing that the algorithm satisfies the following Theorem [9, page 8], which essentially states that the HOPE algorithms finalize precisely those intervals which have been definitely affirmed:

Theorem 5.1 *For all intervals B,* **finalize(B)** *occurs iff* **affirm(X)** *is applied to all of the AIDs $X \in B.IDO$ by intervals that eventually become definite.*

Subsection 5.2 describes an algorithm in which the dependency sets are updated without regard to interleaving conflicts. We show that this algorithm satisfies Theorem 5.1 *if* the intervals executing concurrent **affirm** primitives do not mutually depend on the AIDs being affirmed by the other intervals, then the algorithm detects and corrects for conflicts. Subsection 5.3 extends this algorithm so that it satisfies Theorem 5.1 under all circumstances.

5.2 A BASIC ALGORITHM

This section describes **Algorithm 1**: an algorithm for HOPE that implements the set updates, but does not prevent interleaving conflicts. The algorithm keeps user programs free of synchronization delay because at no point in the execution of a HOPE primitive does any user process wait for acknowledgment from any other process.

Processes make updates to the dependency sets and histories of remote processes by sending messages. Table 1 lists the message types, the source and destination of the message, the contents, and the meaning of the message. The source and destination specification "User" refers to the HOPE modules attached to user processes as in Figure 3, and "AID" refers to an AID process.

A message is denoted $< Type, iid, IDO >$. Some messages omit some arguments, which are considered $\emptyset$. The iid field is the identity of either the sending or destination interval. The IDO set is either the IDO set of the sending interval in the user process, or the set intended to replace the sending AID in the destination interval's IDO set.

Each HOPE primitive is provided as a function that takes an AID as its argument. Primitive execution manipulates the local sets, sends a message to the specified AID process, and possibly messages to other AID processes in the interval's dependency sets.

All of the primitives except **guess** expect the argument to be the process identifier of an AID process. **guess** will also use an AID as an argument, but in addition, if the argument is $\emptyset$, then **guess** infers that this is a *new* optimistic assumption and spawns a new AID process to track the new optimistic assumption.

AID PROCESSES

An AID process models an AID by storing its state. The **affirm** and **deny** primitives applied to AIDs send Affirm and Deny messages to the AID process. The remaining message types do the dependency tracking bookkeeping. Execution of the AID processes is described using state machines that loop forever processing messages.

An AID process responds to a message according to the message contents and the state of the AID. The AID state is represented by the dependency sets, and the variable **state** which represents the following possible truth value of the associated optimistic assumption. There are three additional truth values in addition to **True** and **False** to reflect the partial knowledge that optimism introduces:

Cold	the AID has not had any primitives applied to it yet
Hot	AID has received a Guess message, but has not yet been affirmed
Maybe	AID has received an Affirm message, but was affirmed subject to the set IDO of other AID's also being affirmed
True	AID has been unconditionally affirmed
False	AID has been unconditionally denied

AID processes record the following dependency sets:

DOM	Depends On Me set of intervals contingent on this AID
A_IDO	Affirm–I Depend On set of AIDs that predicate the **affirm** of this AID

The AID state machine begins in state **Cold**, and "terminates" in state **True** or **False**, which are final states. The AID process does not terminate, however, because there may still exist processes with pending **guess** primitives to be applied to the AID, so the AID process must continue to respond to **guess** messages. Reference counting can garbage collect old AID processes.

Figure 4 shows the major state transitions of the AID state machine. Figure 5 shows the top level of the formal specification of the AID state machine. The machine receives messages, and uses the type of the message to select further processing, as shown in Figures 6, 7, and 8. The following text informally describes what the algorithm is doing. The comments throughout assume that X is the identity of the assumption associated with this AID process, and that the variable **my_pid** will reflect this as P_X. The variable **sender** indicates the process identifier of the process that sent the message to P_X.

Guess Message Processing: **Guess** eventually returns either "true" or "false," indicating the final state of the AID. Thus Guess messages are requests from User processes to AID processes for the terminal state of the AID process: either **True** or **False**. If P_X is in state **Cold** or **Hot**, then the terminal state of the AID is not yet known, and so the AID process adds the sender to the $P_X.DOM$ set until the state is resolved into either **True** or **False**. If P_X is in state **Maybe**, then it has been **speculatively affirmed;** the validity of the affirmation is dependent on all of the other AIDs

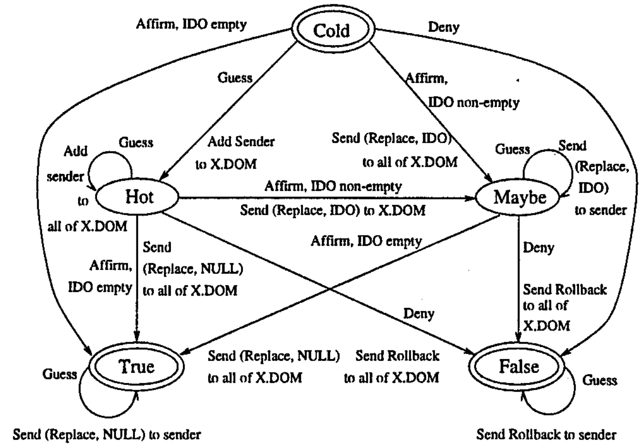

Figure 4: AID State Machine Diagram

```
state := Cold
for ever do
    M := receive any message

    switch M.type:
    case Guess:
        process_guess()
    case Affirm:
        process_affirm()
    case Deny:   // unconditional
        process_deny()
    end switch
end for
```

Figure 5: AID State Machine for P_X

in the $P_X.A\_IDO$ set (the set used to speculatively affirm X) also being affirmed. So P_X sends a Replace message back to the sender, telling the sender to depend on the list of AID processes specified in $P_X.A\_IDO$ set instead of X (P_X in effect "passes the buck"). Finally, if the AID process is in state **True** or **False**, the request can be answered immediately and the AID process sends back a $< Replace, A, \emptyset >$ (replace X with $\emptyset$ in $A.IDO$) or $< Rollback, A >$ message, respectively.

Affirm Message Processing: An Affirm message asserts that the AID's assumption is true. If the $M.IDO$ set is empty, then the AID process P_X has been **definitely affirmed,** and so proceeds directly to state **True**, and sends $< Replace, A, \emptyset >$ messages to all intervals A found in the $P_X.DOM$ set. Otherwise the assertion is tentative and depends on all of the other AID processes in the $M.IDO$ set also being affirmed, and so sets $A\_IDO = M.IDO$ and **state** to **Maybe**, and sends $< Replace, A, A\_IDO >$ messages to all intervals A found in the $P_X.DOM$ set.

Deny Message Processing: Unlike Affirm messages, Deny messages are always unconditional[1]. AID processes in states **True** and **False** ignore Deny messages; in all other states the AID process unconditionally proceeds to state **False**, and sends Rollback messages to all processes who's intervals appear in $P_X.DOM$.

[1] Deny primitives can be buffered until they are definite.

```
process_guess()
    switch state:
    case Cold:
        DOM := {sender} // record the Guess
        state := Hot

    case Hot:
        DOM := DOM ∪ {sender} // record the Guess
        // state is unchanged

    case Maybe:
        send <Replace, M.iid, A_IDO> to sender
        // tells the sender to depend on the list of
        // AID processes specified in the A_IDO set
        // instead of X
        // state is unchanged

    case True:
        send <Replace, M.iid, ∅> to sender
        // replace X with ∅ in sender's IDO
        // state is unchanged

    case False:
        send <Rollback, M.iid> to sender
        // state is unchanged
    end switch
end process_guess
```

Figure 6: Guess Message Processing

```
process_affirm()
    switch state:

    case Cold, Hot, Maybe:
        A_IDO := M.IDO
        for all members B ∈ DOM set do
            send <Replace, B.iid, A_IDO> to B.pid
        end for
        if A_IDO = ∅ then
            state := True
        else
            state := Maybe

    case True, False:    // user error
        abort

    end switch
end process_affirm
```

Figure 7: Affirm Message Processing

CONTROL MESSAGE PROCESSING IN USER PROCESSES

Dependency tracking requires changing the execution history of user processes executing on remote machines. The AID processes make these changes by sending messages back to user processes, which are intercepted and applied by the function Control in `HOPElib` (see Figure 3). Control treats the sequence of intervals in the process's history as a set of state machines, using the message contents and the state of the specified interval to compute the required updates. An interval state is comprised of the following:

IDO	I Depend On
IHD	I Have Denied
IHA	I Have Affirmed

Figure 9 shows an interval state machine, Figure 10 shows the formal definition of the control state machine, and the following text presents an informal description. **Sender** is the sending AID, and **target** is the interval that the message should be applied to. As mentioned earlier, all Control operations are completely transparent to the programmer.

```
process_deny()
    switch state:

    case Cold, Hot, Maybe:
        for all members B ∈ DOM set do
            send <Rollback, B.iid> to B.pid
        end for
        state := False

    case False: // redundant, ignore

    case True:    // user error
        abort

    end switch
end process_deny
```

Figure 8: Deny Message Processing

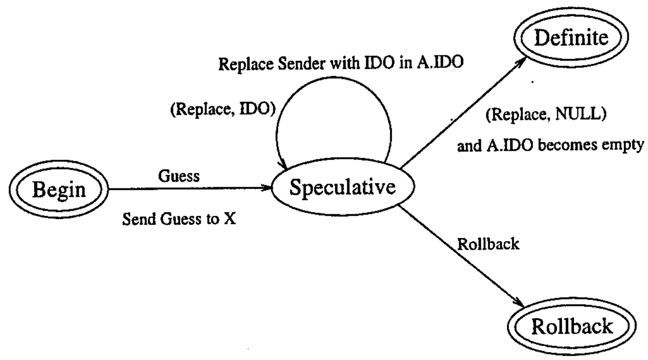

Figure 9: Control: Interval State Machine Diagram

Rollback Message Processing: A Rollback message causes the specified interval A, and all subsequent intervals, to be rolled back. The rollback happens regardless of the state of A, so long as A has not already been rolled back. The process is rolled back to the state immediately preceding the beginning of interval A.

Replace Message Processing: Replace indicates that the sending AID process P_X should be removed from the IDO set of the specified interval A, and replaced with the accompanying $M.IDO$ set. If the resultant $A.IDO$ set is empty, then interval A is finalized.

If the interval is not finalized, then it must update its dependencies. [9] specifies that speculative execution of **affirm(X)** in interval A should add all intervals listed in $X.DOM$ to the DOM set

```
control (message M)
    target := M.iid

    switch M.type
    case Rollback:
        if target ∈ history then
            rollback(target)
        end if

    case Replace:
        if M.IDO = ∅ then
            target.IDO := target.IDO \ {sender}
            if target.IDO = ∅ then
                finalize(target)
            end if
        else if M.IDO ≠ ∅ then
            for each Y ∈ M.IDO do
                target.IDO := (target.IDO ∪ {Y})
                               \ {sender}
                send <Guess, target> to P_Y
            end for
        end if
    end switch
end control
```

Figure 10: Control State Machine

of all AIDs listed in $A.IDO$. The AID process initiated this addition to the DOM sets by sending the Replace message to all dependent intervals, and the Control function completes the DOM addition by sending Guess messages to all of the new AID processes in the replacement IDO set to add the sending interval to the DOM set of the receiving AID process.

Control applies **finalize** and **rollback** functions to intervals in a history. Finalize(A) causes A to become definite, and makes appropriate updates to AID processes that were the subject of speculative HOPE primitives within interval A. Rollback(A) similarly rolls back the interval A, and applies updates to AID processes that were the subject of speculative HOPE primitives within interval A. Figure 11 presents the algorithms for finalize and rollback.

Satisfying Theorem 5.1 We now show that Algorithm 1 satisfies Theorem 5.1 under certain circumstances. We specify the circumstances using a **dependency graph** of the dependencies between intervals and AIDs. Nodes of the graph are intervals and AIDs, and edges represent the "depends on" relation:

Definition 5.1 *An interval A* **depends on** *an AID Y when ever $Y \in A.IDO$, denoted as $A \rightarrow Y$.*

Definition 5.2 *An AID X* **depends on** *an AID Y when ever $Y \in P_X.A\_IDO$, denoted as $X \rightarrow Y$.*

Consider a circumstance in which interval A depends on AID Y, and interval B depends on AID X; **affirm(X)** is executed in A and **affirm(Y)** is executed in interval B. Figure 12 shows the dependency graph sequence in the non-interleaved case where **affirm(X)** is executed first. The speculative **affirm(X)** in A while A depends on Y adds Y to $P_X.A\_IDO$, represented by the edge $X \rightarrow Y$.

Figure 13 shows the dependency graph sequence when the execution of **affirm** primitives interleaved and interfere. The speculative **affirm(X)** in A while A depends on Y introduces a dependency from $X \rightarrow Y$, as before. However, the simultaneous speculative **affirm(Y)** in B while B depends on X also introduces a

```
finalize (interval A)
    remove the checkpoint of the process state created
        when A was started
    mark A as "definite'' in the process history
    for all members Y ∈ A.IHA do
        // unconditional Affirm
        send <Affirm, A, ∅> to P_Y
    end for
    for all members Y ∈ A.IHD do
        send <Deny> to P_Y
    end for
end finalize

rollback (interval A)
    for all members Y ∈ A.IHA do
        send <Deny> to P_Y
    end for
    roll back the process to the state checkpointed at
        the beginning of interval A
    truncate the process history just prior to A
    return False to the guess primitive that initiated
        interval A
end rollback
```

Figure 11: Interval Management Functions

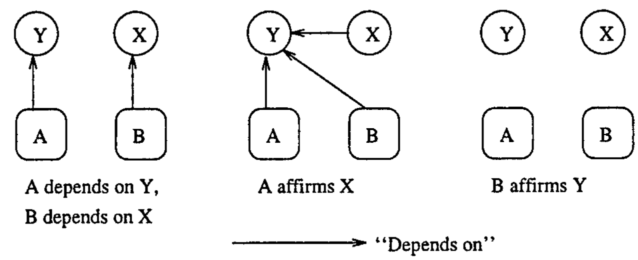

Figure 12: Non-interleaved Affirm Dependency Graph

dependency from $Y \rightarrow X$. Thus a **cyclic dependency** is formed between Y and X. Dependency cycles can occur in rings of any size.

We now show that Algorithm 1 satisfies Theorem 5.1 for acyclic dependency graphs. Lemma 5.1 shows that Algorithm 1 corrects for conflicts between concurrent affirms if the resultant dependency graphs are acyclic, and Lemma 5.2 similarly shows that Algorithm 1 corrects conflicts between concurrent affirms and guesses. Lemma 5.3 shows that a speculative **affirm** in an interval that is later finalized has the same effect as a definite **affirm**. Lemma 5.4 shows that if all AIDs that an interval A depends on are definitely affirmed, then A will be finalized. Finally, Theorem 5.2 shows that Algorithm 1 satisfies Theorem 5.1 if the program's execution forms dependency graphs that are always acyclic.

Lemma 5.1 *For any two conflicting affirms, either:*

1. *The conflicting operations commute to produce the same result,*
2. *Algorithm 1 corrects for the conflict, or*
3. *A cyclic dependency is formed.*

Proof: A construction that shows that for all possible forms of conflict, Algorithm 1 meets one of the criteria [7, pages 67-73]. First

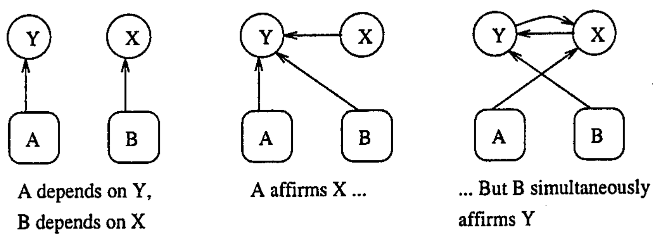

Figure 13: Interleaved Affirm Dependency Graphs: Interference

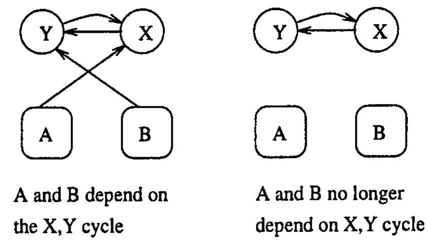

Figure 14: Correcting Cyclic Dependency

the set of atomic read and write operations resulting from **affirm** executions is identified. Then the construction exhaustively shows that for all of the potential conflicts between these read and write operations, they satisfy one of the three conditions in the lemma.

Lemma 5.2 *For any conflicting concurrent execution of an* **affirm** *primitive and a* **guess** *primitive, either:*

1. *The conflicting operations commute to produce the same result, or*
2. *Algorithm 1 corrects for the conflict.*

Proof: A similar construction to Lemma 5.1 [7, pages 73-75].

Lemma 5.3 Affirm Transitivity: *Let B be an interval in process Q that depends on an AID X, i.e., $X \in B.IDO$, and let all dependency graphs produced by this execution be acyclic. The effect of executing* **affirm(X)** *within a speculative interval A upon $B.IDO$ and the state of P_X, followed by A eventually being finalized, is the same as the effect of definite execution of* **affirm(X)**.

Proof: Definite execution of **affirm(X)** will place P_X in state **True** and send $< Replace, B, \emptyset >$ to process Q, removing X from $B.IDO$ and to placing P_X in state **True**.

Let speculative interval A execute **affirm(X)** for some $X \in B.IDO$. Speculative **affirm(X)** in A places P_X in state **Maybe**, and sets $P_X.A\_IDO \leftarrow A.IDO$ (Figure 7). Since $B \in P_X.DOM$, P_X will send $< Replace, B, A\_IDO >$ to process Q. Control uses this message to replace X with $A.IDO$ in $B.IDO$, and sends $< Guess \ldots >$ messages to all AID processes in $A.IDO$ (Figure 10). The $< Guess \ldots >$ messages add B to the DOM set attached to each AID process in $A.IDO$, ensuring that B is in the DOM set of all AID processes that also contain A. Let $\alpha = A.IDO$ be the set of AIDs that replace X.

By assumption, interval A is subsequently finalized. Since Control will only finalize A if $A.IDO = \emptyset$, all AID processes in $A.IDO$ have entered state **True**, and thus have been replaced with $\emptyset$. Since B is in all of the DOM sets that contain A, all Replace messages sent to A will also be sent to B. Lemma 5.1 and the acyclic assumption assure that any concurrency conflicts between Replace messages are detected and corrected. All replacements made in $A.IDO$ will also be made in $B.IDO$, so the replacements that reduced $A.IDO$ to $\emptyset$ will remove all of α from $B.IDO$. Thus X has effectively been removed from $B.IDO$.

Since A has been finalized, a $< Affirm, A, \emptyset >$ message is sent to X (Figure 11), placing X in state **True** (Figure 7). Thus the effect is the same as a definite **affirm(X)**. □

Lemma 5.4 *For any interval B, if* **affirm** *is definitely executed on all AIDs in $B.IDO$, then* **finalize(B)** *results.*

Proof: Definite **affirm** on each $X \in B.IDO$ sends $< Affirm, \ldots, \emptyset >$ to each P_X, placing each P_X in state **True** and sends $< Replace, \ldots, \emptyset >$ messages to each interval in $P_X.DOM$ (Figure 7).

Each X was added to $B.IDO$ by a **guess** primitive or a Replace message; in both cases $< Guess \ldots >$ is sent to the associated AID process P_X to add B to $P_X.DOM$. Lemma 5.2 assures that concurrency conflicts between Replace and Guess messages are detected and corrected. Thus for each $X \in B.IDO$, it is also the case that $B \in P_X.DOM$.

Each AID process P_X has sent $< Replace, \ldots, \emptyset >$ to each interval in its DOM set, including all intervals in $B.IDO$. Thus all AIDs listed in $B.IDO$ have been replaced with $\emptyset$, inducing Control to finalize B (Figure 10). □

Theorem 5.2 *For all intervals B in any execution where the dependency graph is always acyclic, Algorithm 1 executes* **finalize(B)** *iff* **affirm(X)** *is applied to all AIDs $X \in B.IDO$ by intervals that are eventually finalized.*

Proof: First we show that if **affirm(X)** is applied to all AIDs $X \in B.IDO$ by intervals that eventually become definite, then **finalize(B)** will result.

Lemma 5.4 shows that if all of the **affirm(X)** primitives are definitely executed, then **finalize(B)** will result. Lemma 5.3 shows that speculatively executing **affirm** in intervals that are eventually finalized has the same effect as definite affirms, and so **finalize(B)** results in either case.

Assume that B has been finalized, and that $\exists X \in B.IDO$ that has not been affirmed, and use contradiction to prove that **finalize(B)** implies that **affirm(X)** has been applied to all AIDs $X \in B.IDO$. Since X has not been affirmed, no operation will have removed X from $B.IDO$. Therefore $B.IDO \neq \emptyset$, preventing Control from finalizing B, contradicting the assumption that B has been finalized. Therefore, for all intervals B, B is finalized *iff* all of the AIDs $X \in B.IDO$ are affirmed in intervals that eventually become definite. □

5.3 CYCLE DETECTION

Algorithm 1 satisfies Theorem 5.1 for acyclic dependency graphs. However, if the dependency graph becomes cyclic, as in Figure 13 when mutual **affirm** primitives are executed simultaneously, then Algorithm 1 will fail to detect the cycle, and the participating intervals will "bounce" their way around the cyclic of dependent AIDs forever [7].

Algorithm 2 extends Algorithm 1 to detect and remove dependencies from intervals to AIDs that are members of a cycle. Figure 14 shows the dependency graph progression from the cyclic dependency in Figure 13 to a state in which the intervals no longer depend on the cycle. If one or more of the intervals that executed

```
control (message M)
    target := M.iid

    switch M.type
    case Rollback:
        if target ∈ history then
            rollback(target)
        end if

    case Replace:
        if M.IDO = ∅ then
            target.IDO := target.IDO \ {sender}
            if target.IDO = ∅ then
                finalize(target)
            end if
        else if M.IDO ≠ ∅ then
            for each Y ∈ M.IDO do
                if Y ∈ target.UDO then
                    target.IDO := target.IDO \ {sender}
                    if target.IDO = ∅ then
                        finalize(target)
                    end if
                else
                    target.IDO := target.IDO ∪ {Y}
                    send <Guess, target> to P_Y
                end if
            end for
            target.IDO := target.IDO \ {sender}
            target.UDO := target.UDO ∪ {sender}
        end if
    end switch
end control
```

Figure 15: Control State Machine with Cycle Detection

the speculative affirms that constructed the cycle are eventually finalized as a result, they will send unconditional Affirm messages to the members of the cycle, causing them to be definitely affirmed. The collection of dependency sets stored with each interval is extended with a **UDO** (Used to Depend On) set of AIDs that used to be in IDO, to prevent a process from cyclicly exchanging one dependency for another.

The extended Control function in Figure 15 implements cycle detection. In addition to removing the sender of the Replace message from the specified interval's IDO set, the sender is also added to the UDO set. If a Replace message arrives with a non-empty replacement $M.IDO$ set, then $M.IDO$ is compared against the specified interval's UDO set. If an AID in $M.IDO$ is found in the UDO set, it is discarded: This represents a cycle in the dependency loop, and once detected the interval no longer needs to depend on the cycle. We show that this completely satisfies Theorem 5.1 by showing that it removes all dependencies from intervals to AID nodes that are members of a cycle.

Lemma 5.5 *All members of a dependency cycle must be AIDs that have been speculatively affirmed.*

Proof: We eliminate all nodes from the graph other than AID nodes that have been speculatively affirmed. We first note that all nodes in a cycle must have both in and out edges, and then show that only speculative affirms can attach both in and out edges to an AID node.

Interval nodes only have out-bound dependency edges, because no other node can ever depend on an interval. Thus dependency cycles can only be composed of AIDs. The A_IDO set associated with each AID process defines the set of out-bound edges attached to each AID node. Only the speculative execution of **affirm(X)** will set $P_X.A\_IDO$ to a non-null value, as shown in Figure 7. Thus all members of a cyclic dependency must be AID nodes that have been speculatively affirmed. □

Theorem 5.3 *If a set of AID processes forms a dependency graph G that contains a cycle C, then Algorithm 2 will remove all dependencies from speculative intervals on all members of the set C.*

Proof: By inspection, we show that speculative **affirm** processing detects and eliminates dependencies on members of cycles. An AID process P_X that has been speculatively affirmed is left in state **Maybe** with an A_IDO set indicating the list of other AIDs that it depends on from the speculative affirm. An interval A in a user process attempting to depend on X by sending it a Guess message (either from a user **guess** primitive or from processing a Replace message) will get a $< Replace, A, A\_IDO >$ message as a result. A_IDO contains the set of AIDs that X depends on. Thus user processes that attempt to depend AIDs that have been speculatively affirmed are forced to instead depend on the set of AIDs that the speculatively affirmed AIDs depend on.

If an AID X is in a dependency cycle, then any interval attempting to depend on X will be forced to instead depend on the "next" AID in X's cycle, Y. Attempting to depend on Y will similarly pass the interval on to the following AID, in effect "walking around" the ring of dependencies.

As interval A walks around the dependency cycle, it records the list of AID nodes that it has attempted to depend on in $A.UDO$. When A attempts to depend on an AID node that it has already tried to depend on, it is detected by comparison with $A.UDO$ (see Figure 15). Control responds by deleting A's dependency on the ring. □

6 CONCLUSIONS & FUTURE RESEARCH

We presented a new model for expressing optimism by providing primitives to identify optimistic assumptions, and then later assert whether the assumptions were correct, while automating the dependency tracking necessary to maintain consistency. The model has been implemented, and the algorithm has been shown to be both wait-free, and consistent with the formal semantics of the HOPE primitives with respect to finalizing speculative computation.

Elsewhere we have defined a formal semantics for HOPE [9]. The prototype implementation is freely available [8] and we have shown that the prototype can improve RPC performance by of up to 70%. We have also done preliminary investigations into the application of HOPE to replication [5] scientific programming [6], and software fault tolerance [18]. In future work, we will show that these algorithms are quadratic in the number of intervals and AIDs associated with an `affirm`[2], and we will also extend the application of optimism beyond its traditional domains into new areas such as optimistic specialization [19] and truth maintenance systems [12].

7 ACKNOWLEDGMENTS

HOPE was inspired by optimism studies at the IBM T.J. Watson Research Center by Rob Strom et al. Thanks go to Andy Lowry, Jim Russell and Ajei Gopal of IBM for their early comments on

[2] We expect the N to be small.

this work. Thanks go to Mike Bennett and Andrew Marshall of UWO for ongoing encouragement and commentary, and to Roger Barga, Jonathan Walpole, and Calton Pu of OGI for their comments on this paper.

REFERENCES

[1] David F. Bacon and Robert E. Strom. Optimistic Parallelization of Communicating Sequential Processes. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[2] Rajive L. Bagrodia and Wen-Toh Liao. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, 20(4):225–238, April 1994.

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[4] R.G. Bubenik. *Optimistic Computation*. PhD thesis, Rice University, May 1990.

[5] Crispin Cowan. Optimistic Replication in HOPE. In *Proceedings of the 1992 CAS Conference*, pages 269–282, Toronto, Ontario, November 1992.

[6] Crispin Cowan. Optimistic Programming in PVM. In *Proceedings of the 2nd PVM User's Group Meeting*, Oak Ridge, TN, May 1994.

[7] Crispin Cowan. *A Programming Model for Optimism*. PhD thesis, University of Western Ontario, March 1995.

[8] Crispin Cowan. HOPE: Hopefully Optimistic Programming Environment. Prototype implementation avialable via FTP from `ftp://ftp.csd.uwo.ca/pub/src/hope.tar.gz`, August 1995.

[9] Crispin Cowan and Hanan Lutfiyya. Formal Semantics for Expressing Optimism: The Meaning of HOPE. In *1995 Symposium on the Principles of Distributed Computing (PODC)*, Ottawa, Ontario, August 1995.

[10] Crispin Cowan, Hanan Lutfiyya, and Mike Bauer. Increasing Concurrency Through Optimism: A Reason for HOPE. In *Proceedings of the 1994 ACM Computer Science Conference*, pages 218–225, Phoenix, Arizona, March 1994.

[11] Crispin Cowan, Hanan Lutfiyya, and Mike Bauer. Performance Benefits of Optimistic Programming: A Measure of HOPE. In *Fourth IEEE International Symposium on High-Performance Distributed Computing (HPDC-4)*, August 1995.

[12] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12:231–272, 1979.

[13] Al Geist, Adam Geguelin, Jack Dongarra, Wicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine, a Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1995.

[14] D. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 3(7):404–425, July 1985.

[15] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems using Optimistic Message Logging and Checkpointing. *J. Algorithms*, 11(3):462–491, September 1990.

[16] Puneet Kumar. Coping with Conflicts in an Optimistically Replicated File System. In *1990 Workshop on the Management of Replicated Data*, pages 60–64, Houston, TX, November 1990.

[17] H.T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[18] Hanan Lutfiyya and Crispin Cowan. Language Support for the Application-Oriented Fault Tolerance Paradigm. Submitted for review, 1995.

[19] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[20] R.E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[21] P. Triantafillou and D.J. Taylor. A New Paradigm for High Availability and Efficiency in Replicated and Distributed Databases. In *2nd IEEE Symposium on Parallel and Distributed Processing*, pages 136–143, December 1990.